Milan House Prices Forecasting Data Challenge

1. Load the Data

Train and test datasets are provided in .csv format. Data were loaded using pandas' `'read_csv'` function and then concatenated together with the `'concat'` function.

2. Data processing

The first to be noticed is that the 'other_features' column contains important information to be extracted. Each entry is a string containing several additional features of the house, separated by the '|' character. Using Regular Expressions I was able to separate each single feature. After some additional cleaning of the strings, I obtained 40 unique features, which were added as dummy variables to the data frame. For instance, a new column like 'security door' was introduced, which is set to True if a house has a security door and False otherwise. After this manipulation, the dataset has a total of 57 columns. I proceeded by analyzing and manipulating each feature separately.

Bathrooms_number

The range of bathrooms varies from 1 to 3 or more, with most houses containing just 1 bathroom. To utilize this data as an ordinal variable, I converted text to integer values. Specifically, I replaced entries denoted as '3+' with the integer 5, assuming it likely reflects the average number of bathrooms for houses with more than 3.

Rooms_number

Same as above, I wanted to use this as an ordinal variable. Here, I replaced '5+' entries with 7, following the same reasoning used for bathrooms.

Car_parking

In this case, I wanted to reduce the number of categories. To do so, I grouped entries in the following way:

- 'garage': houses that have 1 spot in the garage/box
- 'shared': houses with 1 or more shared spots but no spots in the garage/box
- 'more_garage': house with more than 1 garage/box spot but no shared parking
- 'shared_and_garage': houses with 1 spot in the garage/box, and at least 1 shared parking
- 'more_shared_and_garage': houses with more than 1 spot in the garage/box, and at least 1 shared parking
- 'No': house with no parking at all

Total_floors_in_building

The number of total floors in the building ranges from 1 to 27. I processed the data to obtain the following groups:

- 0-2 floors
- 3-4 floors
- 5-6 floors
- 7-9 floors
- more than 9 floors ('10+')

Choices about groupings are made based on different characteristics of a house that can reasonably impact its final price. In addition, I tried not to have very unbalanced or unrepresented groups.

Energy_efficiency_class

This variable spans from 'a' (best efficiency class) to 'g' (worst efficiency class). I transformed it into an ordinal variable assigning integer values from 1 ('g') to 7 ('a'). Additionally, I replaced two entries denoted by ',' in the original dataset with the `np.nan()` special value.

Condominium_fees

There were 393 different values in the condominium_fees column. Again, I wanted it to be an ordinal feature. In my transformed column, values range from 0 (no fees) to 10 (more than 900 fees) with a step of 100 between each group.

Availability

As I did not want to lose information from the availability feature, but the vast majority of houses were already available, I considered all houses available from 2023 to 2024 as 'available soon' and from 2024 onward as 'available later'. I labeled all other houses as 'available'.

Floor

I deemed it reasonable to consider this variable as categorical, as floors should not have an underlying ranking. Hence, no processing was applied to this feature.

Heating_centralized & Conditions

These two variables were left as categorical, following the same reasoning as for floor number.

Year_of_construction

The year of construction was originally a numerical feature. I transformed it into a categorical variable by collecting houses constructed in the same decade. Here are the final groups:

- Pre-1900
- 1900-1920
- 1920-1930
- 1930-1940
- 1940-1950
- 1950-1960
- 1960-1970
- 1970-1980
- 1980-1990
- 1990-2000
- Post-2000

By categorizing years into decades, the model could learn distinct effects associated with different time periods, which may not be captured effectively by treating the year as a continuous variable.

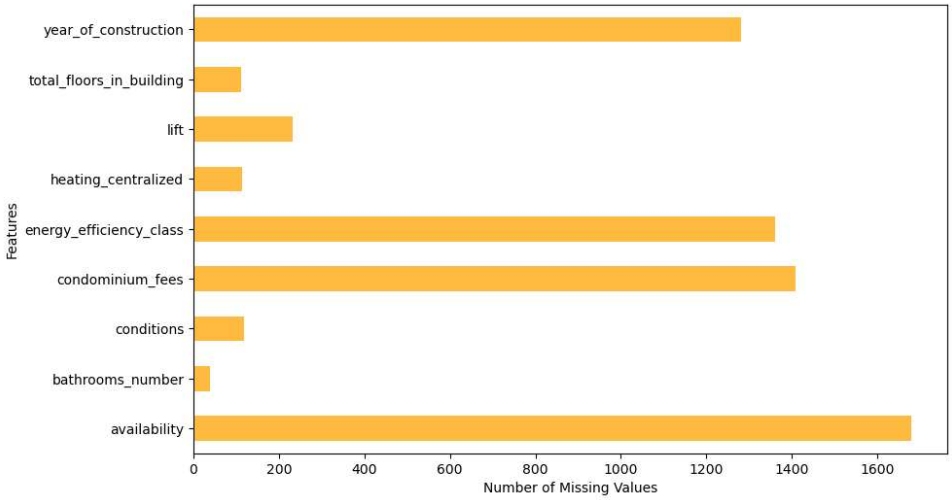
Secondly, this transformation can mitigate the influence of outliers or noise associated with individual years.

Zone

The feature related to the zone of the houses was the most complicated to deal with. In the original dataset, the division in zones was very granular, leading to many zones having very few observations. To tackle this issue, I opted to group zones with fewer than 30 observations together based on their similarities in location and average prices as observed on *'Immobiliare.it'*. Through this manipulation, I reduced the total number of zones from 149 to 123.

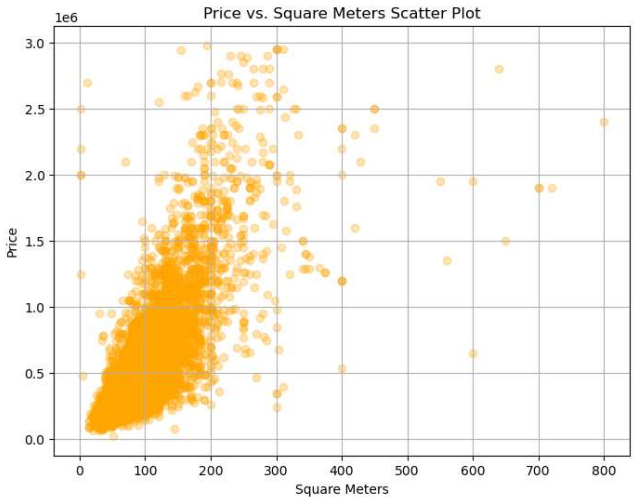
3. Missing Values

The strategy I followed was to substitute missing values with the mode of the feature and add a variable indicating if the entry for a specific feature was missing. After doing this, I am left with 70 columns in my dataset (considering weights and prices).



4. Feature engineering

I decided to add the square and the cube of the square meters variable to get more information from one of the most important features in determining the house price, at least from a theoretical perspective. For the same reason, I built a dummy variable indicating if the house is above 220 square meters. The scatter plot depicts a positive correlation between prices and square meters.



Furthermore, I created a variable capturing the average room size in the house. I did this by dividing the square meters by the sum of bathrooms number and rooms number. These decisions are mainly based on theoretical assumptions about the definition of house prices.

5. Encoding

In this step, I simply encoded categorical variables by using the `pd.get_dummies()` function. The `drop_first` parameter is set to True to avoid multicollinearity.

6. Algorithm

First, I created a validation set using 20% of the observations in the training set to have a reliable assessment of my algorithms.

I chose to implement the *Extreme Gradient Boosting Regressor* (XGBRegressor from the XGBoost library) for my predictive modeling tasks. *XGBoost* is known for its efficient implementation of the gradient boosting algorithm, making it a popular choice for achieving high-performance predictions. Initially, I selected this algorithm based on its reputation for delivering strong results across a variety of prediction tasks. To validate my choice, I conducted comparative analyses against other algorithms like *Random Forest* and *AdaBoost Regressor*. Through this evaluation, I observed that XGBoost consistently outperformed these alternatives. Therefore, I determined that XGBoost was the optimal solution for this specific task and I proceeded to fine-tune its parameters using cross-validation.

I performed hyperparameter tuning for the *XGBoost regressor* model using grid search cross-validation. The `param_grid` dictionary specifies a grid of hyperparameters that I wanted to explore, including different values for `n_estimators` (number of boosting rounds), `max_depth` (maximum depth of each tree), and `learning_rate` (step size shrinkage). I instantiated the model with fixed parameters such as `min_child_weight`, `colsample_bytree`, `colsample_bylevel`, `colsample_bynode`, and `subsample` which are not included in the grid search.

The subsample ratio in *XGBoost* indicates the percentage of training instances used for training the base learner at each boosting iteration. I chose 0.6 after some empirical testing.

The grid search process fits the model with each combination of hyperparameters and evaluates its performance using 5-fold cross-validation. The scoring metric used is the negative mean absolute error (`scoring='neg_mean_absolute_error'`), where a lower value indicates better performance.

After the grid search is complete, the best combination of hyperparameters (`best_params_`) is printed, representing the configuration that yielded the lowest mean absolute error during cross-validation.

The choice to fit the model with the logarithm of the dependent variable (`np.log(y_values_train_for_valid)`) followed by transforming predictions back to the original scale using the exponential function (`np.exp(y_pred_valid)`) is motivated by the willingness to reduce the impact of extreme values during model training, thus enhancing the stability and performance of the model.

The Mean Absolute Error between the true validation target values and the transformed predictions provides the measure of prediction accuracy in the validation dataset.

The model is then fitted on the entire training set (i.e. including those observations used as validation set) and the predictions are used as the final submission.

7. Results

To gain some insights about the model, I looked at feature importance. Feature importance helps in identifying which features have the most significant impact on the model's predictions. Specifically, in XGBoost, the importance of a feature is computed as the number of times it appears in all the splitting decisions across all the trees in the ensemble. In the table below the top 15 features with the highest importance are listed.

Features	Importance
bathrooms_number	0.03228125721216202
zone_grouped_baggio	0.02587321773171425
square_meters_3	0.025330189615488052
square_meters_2	0.023693779483437538
square_meters	0.02244037389755249
zone_grouped_carrobbio	0.01787274144589901
zone_grouped_quarto oggiaro	0.016896752640604973
zone_grouped_roserio	0.015706242993474007
zone_grouped_vialba	0.01509652566164732
zone_grouped_vincenzo monti	0.014262937940657139
zone_grouped_moscova	0.014252443797886372
zone_grouped_palestro	0.013331261463463306
floor_semi-basement	0.012925341725349426
zone_grouped_quartiere olmi	0.012126527726650238
zone_grouped_muggiano	0.011788669973611832

Bathrooms_number is by far the most important feature, *square_meters* and its transformations are all in the top 5. Other than *floor_semi-basement*, the remaining features are related to the zone. These findings align well with theoretical expectations, as bathrooms, square meters, and zoning are key factors known to strongly influence property valuation.

Furthermore, it is interesting to investigate if there are clear patterns within the significant prediction errors made by the model. To explore this, I filtered the validation set to isolate observations where the

model's prediction deviates substantially from the true value (by more than 100/200’000 in absolute terms), and then examined the characteristics of these outlier observations.

As one could expect, the model tends to exhibit the largest prediction errors for larger houses. To illustrate this, I calculated the average square meters, number of rooms, and number of bathrooms for observations in the validation set and compared them with the averages of observations where the predicted price deviated by more than 100’000 from the true value. These are the results:

```
Average square meters for wrongly predicted houses: 139.5091463414634
Average square meters in validation set: 96.22875
```

```
Average bathrooms number for wrongly predicted houses: 2.0213414634146343
Average bathrooms number in validation set: 1.463125
```

```
Average rooms number for wrongly predicted houses: 3.6554878048780486
Average rooms number in validation set: 2.886875
```

Results are even more extreme when considering errors of more than 200’000:

```
Average square meters for wrongly predicted houses: 175.3153153153153
Average square meters in validation set: 96.22875
```

```
Average bathrooms number for wrongly predicted houses: 2.4324324324324325
Average bathrooms number in validation set: 1.463125
```

```
Average rooms number for wrongly predicted houses: 4.054054054054054
Average rooms number in validation set: 2.886875
```

8. Additional Notes

The code is entirely runnable once the necessary packages are installed. The two cells where parameters tuning is performed might take time to run (approximately one hour each on my machine).